Operating Systems: Internals and Design Principles

Chapter 10 Multiprocessor, Multicore and Real-Time Scheduling

Ninth Edition
By William Stallings

Classifications of Multiprocessor Systems

Loosely coupled or distributed multiprocessor, or cluster

- Consists of a collection of relatively autonomous systems, each processor having its own main memory and I/O channels

Functionally specialized processors

- There is a master, general-purpose processor;
- Specialized processors are controlled by the master processor and provide services to it

Tightly coupled multiprocessor

- Consists of a set of processors that share a common main memory and are under the integrated control of an operating system

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	<20
Medium	Parallel processing or multitasking within a single application	20-200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200-2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000-1M
Independent	Multiple unrelated processes	not applicable

Table 10.1 Synchronization Granularity and Processes

Independent Parallelism

- No explicit synchronization among processes
  - Each represents a separate, independent application or job
- Typical use is in a time-sharing system

Each user is performing a particular application

Multiprocessor provides the same service as a multiprogrammed uniprocessor

Because more than one processor is available, average response time to the users will be shorter

Coarse and Very Coarse Grained Parallelism

- There is synchronization among processes, but at a very gross level
- Easily handled as a set of concurrent processes running on a multiprogrammed uniprocessor
- Can be supported on a multiprocessor with little or no change to user software

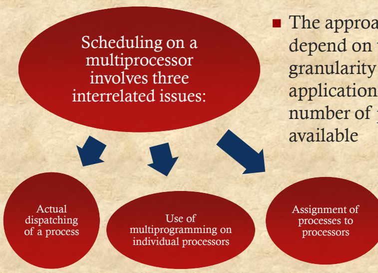
Medium-Grained Parallelism

- Single application can be effectively implemented as a collection of threads within a single process
  - Programmer must explicitly specify the potential parallelism of an application
  - There needs to be a high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization
- Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application

Fine-Grained Parallelism

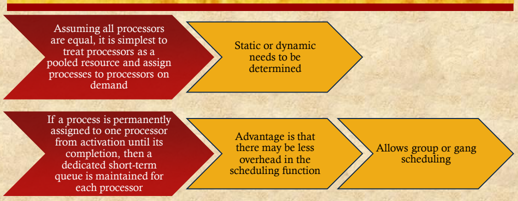
- Represents a much more complex use of parallelism than is found in the use of threads
- Is a specialized and fragmented area with many different approaches

Design Issues



- The approach taken will depend on the degree of granularity of applications and on the number of processors available

Assignment of Processes to Processors



- A disadvantage of static assignment is that one processor can be idle, with an empty queue, while another processor has a backlog
  - To prevent this situation, a common queue can be used
  - Another option is dynamic load balancing

Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Approaches:
  - Master/Slave
  - Peer

Master/Slave Architecture

- Key kernel functions always run on a particular processor
- Master is responsible for scheduling
- Slave sends service request to the master
- Is simple and requires little enhancement to a uniprocessor multiprogramming operating system
- Conflict resolution is simplified because one processor has control of all memory and I/O resources

Disadvantages:

- Failure of master brings down whole system
- Master can become a performance bottleneck

Peer Architecture

- Kernel can execute on any processor
- Each processor does self-scheduling from the pool of available processes

Complicates the operating system

- Operating system must ensure that two processors do not choose the same process and that the processes are not somehow lost from the queue

Process Scheduling

- In most traditional multiprocessor systems, processes are not dedicated to processors
- A single queue is used for all processors
  - If some sort of priority scheme is used, there are multiple queues based on priority, all feeding into the common pool of processors
- System is viewed as being a multi-server queuing architecture

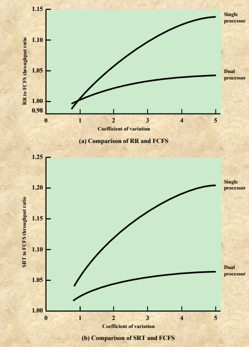


Figure 10.1 Comparison of Scheduling Performance for One and Two Processors

Thread Scheduling

- Thread execution is separated from the rest of the definition of a process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
- On a uniprocessor, threads can be used as a program structuring aid and to overlap I/O with processing
- In a multiprocessor system threads can be used to exploit true parallelism in an application
- Dramatic gains in performance are possible in multi-processor systems
- Small differences in thread management and scheduling can have an impact on applications that require significant interaction among threads

Approaches to Thread Scheduling

Four approaches for multiprocessor thread scheduling and processor assignment are:

Load Sharing

Processes are not assigned to a particular processor

Gang Scheduling

A set of related threads scheduled to run on a set of processors at the same time, on a one-to-one basis

Dedicated Processor Assignment

Provides implicit scheduling defined by the assignment of threads to processors

Dynamic Scheduling

The number of threads in a process can be altered during the course of execution

Load Sharing

- Simplest approach and the one that carries over most directly from a uniprocessor environment

Advantages:

- Load is distributed evenly across the processors, assuring that no processor is idle while work is available to do
- No centralized scheduler required
- The global queue can be organized and accessed using any of the schemes discussed in Chapter 9

- Versions of load sharing:
  - First-come-first-served (FCFS)
  - Smallest number of threads first
  - Preemptive smallest number of threads first

Disadvantages of Load Sharing

- Central queue occupies a region of memory that must be accessed in a manner that enforces mutual exclusion
  - Can lead to bottlenecks
- Preemptive threads are unlikely to resume execution on the same processor
  - Caching can become less efficient
- If all threads are treated as a common pool of threads, it is unlikely that all of the threads of a program will gain access to processors at the same time
  - The process switches involved may seriously compromise performance

Gang Scheduling

- Simultaneous scheduling of the threads that make up a single process

Benefits:
• Synchronization blocking may be reduced, less process switching may be necessary, and performance will increase • Scheduling overhead may be reduced

- Useful for medium-grained to fine-grained parallel applications whose performance severely degrades when any part of the application is not running while other parts are ready to run
- Also beneficial for any parallel application

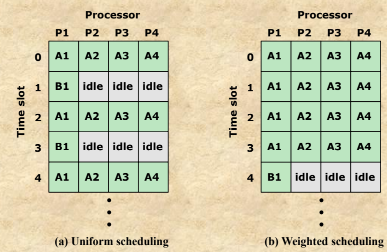


Figure 10.2 Gang Scheduling

Dedicated Processor Assignment

- When an application is scheduled, each of its threads is assigned to a processor that remains dedicated to that thread until the application runs to completion
- If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle
  - There is no multiprogramming of processors
- Defense of this strategy:
  - In a highly parallel system, with tens or hundreds of processors, processor utilization is no longer so important as a metric for effectiveness or performance
  - The total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program

Number of threads per application	Matrix multiplication	FFT
1	1	1
2	1.8	1.8
4	3.8	3.8
8	6.5	6.1
12	5.2	5.1
16	3.9	3.8
20	3.3	3
24	2.8	2.4

Table 10.2 Application Speedup as a Function of Number of Threads

Dynamic Scheduling

- For some applications it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically
  - This would allow the operating system to adjust the load to improve utilization
- Both the operating system and the application are involved in making scheduling decisions
- The scheduling responsibility of the operating system is primarily limited to processor allocation
- This approach is superior to gang scheduling or dedicated processor assignment for applications that can take advantage of it

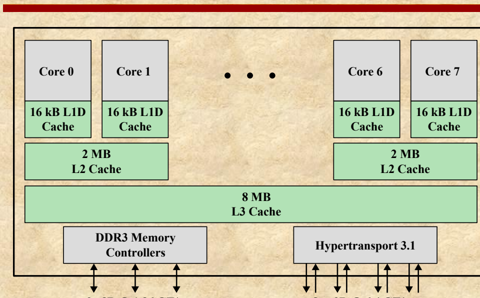


Figure 10.3 AMD Bulldozer Architecture

Cache Sharing

Cooperative resource sharing

- Multiple threads access the same set of main memory locations
- Examples:
  - Applications that are multithreaded
  - Producer-consumer thread interaction

Resource contention

- Threads, if operating on adjacent cores, compete for cache memory locations
- If more of the cache is dynamically allocated to one thread, the competing thread necessarily has less cache space available and thus suffers performance degradation
- Objective of contention-aware scheduling is to allocate threads to cores to maximize the effectiveness of the shared cache memory and minimize the need for off-chip memory accesses

Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component

Examples:
- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
- Tasks or processes attempt to control or react to events that take place in the outside world
- These events occur in "real time" and tasks must be able to keep up with them

Hard and Soft Real-Time Tasks

Hard real-time task

- One that must meet its deadline
- Otherwise it will cause unacceptable damage or a fatal error to the system

Soft real-time task

- Has an associated deadline that is desirable but not mandatory
- It still makes sense to schedule and complete the task even if it has passed its deadline

Periodic and Aperiodic Tasks

- **Periodic tasks**
  - Requirement may be stated as:
    - Once per period *T*
    - Exactly *T* units apart
- **Aperiodic tasks**
  - Has a deadline by which it must finish or start
  - May have a constraint on both start and finish time

Characteristics of Real Time Systems

Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation

Determinism

- Concerned with how long an operating system delays before acknowledging an interrupt
- Operations are performed at fixed, predetermined times or within predetermined time intervals
  - When multiple processes are competing for resources and processor time, no system will be fully deterministic

The extent to which an operating system can deterministically satisfy requests depends on:
- The speed with which it can respond to interrupts
- Whether the system has sufficient capacity to handle all requests within the required time

Responsiveness

- Together with determinism make up the response time to external events
  - Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system
- Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

Responsiveness includes:

- Amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR)
- Amount of time required to perform the ISR
- Effect of interrupt nesting

User Control

- Generally much broader in a real-time operating system than in ordinary operating systems
- It is essential to allow the user fine-grained control over task priority
- User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- May allow user to specify such characteristics as:

Paging or process swapping

What processes must always be resident in main memory

What disk transfer algorithms are to be used

What rights the processes in various priority bands have

Reliability

- More important for real-time systems than non-real time systems
- Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
  - Financial loss
  - Major equipment damage
  - Loss of life

Fail-Soft Operation

- A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible
- Important aspect is stability
  - A real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadlines are not always met
- The following features are common to most real-time OSs
  - A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
  - Interrupt latency is bounded and relatively short
  - More precise and predictable timing characteristics than general purpose OSs

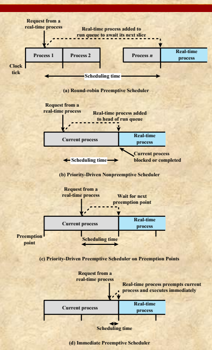


Figure 10.4 Scheduling of Real-Time Process

Real-Time Scheduling

Scheduling approaches depend on:

Whether a system performs schedulability analysis

If it does, whether it is done statically or dynamically

Whether the result of the analysis itself produces a scheduler plan according to which tasks are dispatched at run time

Classes of Real-Time Scheduling Algorithms

Static table-driven approaches

- Performs a static analysis of feasible schedules of dispatching
- Result is a schedule that determines, at run time, when a task must begin execution

Static priority-driven preemptive approaches

- A static analysis is performed but no schedule is drawn up
- Analysis is used to assign priorities to tasks so that a traditional priority-driven preemptive scheduler can be used

Dynamic planning-based approaches

- Feasibility is determined at run time rather than offline prior to the start of execution
- One result of the analysis is a schedule or plan that is used to decide when to dispatch this task

Dynamic best effort approaches

- No feasibility analysis is performed
- System tries to meet all deadlines and aborts any started process whose deadline is missed

Scheduling Algorithms

Static table-driven scheduling

- Applicable to tasks that are periodic
- Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task
- This is a predictable approach but one that is inflexible because any change to any task requirements requires that the schedule be redone
- Earliest-deadline-first or other periodic deadline techniques are typical of this category of scheduling algorithms

Static priority-driven preemptive scheduling

- Makes use of the priority-driven preemptive scheduling mechanism common to most non-real-time multiprogramming systems
- In a non-real-time system, a variety of factors might be used to determine priority
- In a real-time system, priority assignment is related to the time constraints associated with each task
- One example of this approach is the rate monotonic algorithm which assigns static priorities to tasks based on the length of their periods

Scheduling Algorithms

Dynamic planning-based scheduling

- After a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival
- If the new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline, then the schedule is revised to accommodate the new task

Dynamic best effort scheduling

- The approach used by many real-time systems that are currently commercially available
- When a task arrives, the system assigns a priority based on the characteristics of the task
- Some form of deadline scheduling is typically used
- Typically the tasks are aperiodic so no static scheduling analysis is possible
- The major disadvantage of this form of scheduling is, that until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met
- Its advantage is that it is easy to implement

Deadline Scheduling

- Real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching
- Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times
- Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time

Information Used for Deadline Scheduling

Ready time
- Time task becomes ready for execution

Starting deadline
- Time task must begin

Completion deadline
- Time task must be completed

Processing time
- Time required to execute the task to completion

Resource requirements
- Resources required by the task while it is executing

Priority
- Measures relative importance of the task

Subtask scheduler
- A task may be decomposed into a mandatory subtask and an optional subtask

Table 10.3 Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

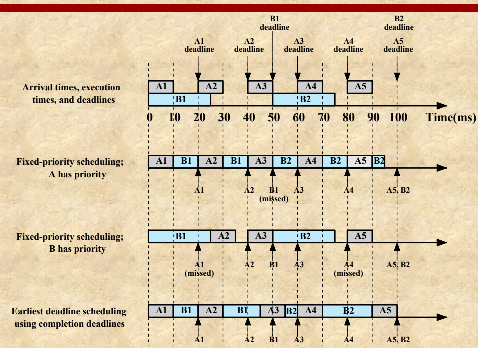


Figure 10.5 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.3)

Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Table 10.4
Execution Profile of Five Aperiodic Tasks

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

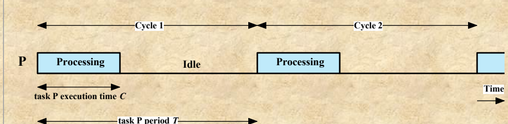


Figure 10.7 Periodic Task Timing Diagram

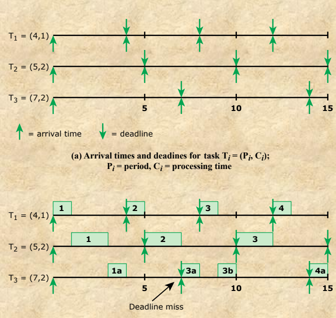


Figure 10.8 Rate Monotonic Scheduling Example

Table 10.5

Value of the RMS Upper Bound

n	$n(2^{1/n} - 1)$
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
•	•
•	•
•	•
¥	ln 2 » 0.693

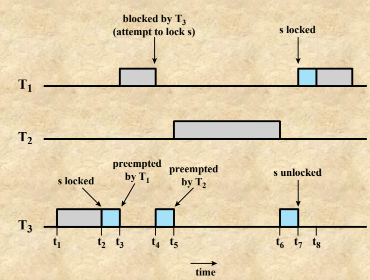
Priority Inversion

- Can occur in any priority-based preemptive scheduling scheme
- Particularly relevant in the context of real-time scheduling
- Best-known instance involved the Mars Pathfinder mission
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task

Unbounded Priority Inversion

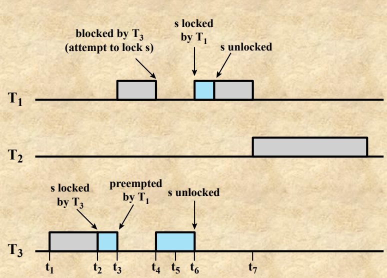
- The duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks

Unbounded Priority Inversion



(a) Unbounded priority inversion

Priority Inheritance



(b) Use of priority inheritance

Linux Scheduling

- The three primary Linux scheduling classes are:
  - SCHED_FIFO: First-in-first-out real-time threads
  - SCHED_RR: Round-robin real-time threads
  - SCHED_NORMAL: Other, non-real-time threads
- Within each class multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_NORMAL class

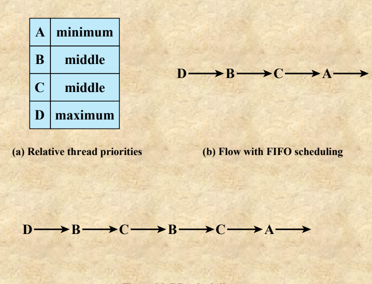


Figure 10.10 Example of Linux Real-Time Scheduling

Non-Real-Time Scheduling

- The Linux 2.4 scheduler for the SCHED_OTHER class did not scale well with increasing number of processors and increasing number of processes
- The drawbacks of this scheduler include:
  - The Linux 2.4 scheduler uses a single runqueue for all processors in a symmetric multiprocessing system (SMP)
    - This means a task can be scheduled on any processor, which can be good for load balancing but bad for memory caches
  - The Linux 2.4 scheduler uses a single runqueue lock
    - Thus, in an SMP system, the act of choosing a task to execute locks out any other processor from manipulating the runqueues, resulting in idle processors awaiting release of the runqueue lock and decreased efficiency
  - Preemption is not possible in the Linux 2.4 scheduler
    - This means that a lower-priority task can execute while a higher-priority task waited for it to complete

Non-Real-Time Scheduling

- Linux 2.6 uses a completely new priority scheduler known as the O(1) scheduler
- The scheduler is designed so the time to select the appropriate process and assign it to a processor is constant regardless of the load on the system or number of processors
- The O(1) scheduler proved to be unwieldy in the kernel because the amount of code is large and the algorithms are complex

Completely Fair Scheduler (CFS)

- Used as a result of the drawbacks of the O(1) scheduler
- Models an ideal multitasking CPU on real hardware that provides fair access to all tasks
- In order to achieve this goal, the CFS maintains a *virtual runtime* for each task
  - The virtual runtime is the amount of time spent executing so far, normalized by the number of runnable processes
  - The smaller a task's virtual runtime is, the higher is its need for the processor
- Includes the concept of sleeper fairness to ensure that tasks that are not currently runnable receive a comparable share of the processor when they eventually need it
- Implemented by the *fair_sched_class* scheduler class

Red Black Tree

- The CFS scheduler is based on using a Red Black tree, as opposed to other schedulers, which are typically based on run queues
  - This scheme provides high efficiency in inserting, deleting, and searching tasks, due to its O(log N) complexity
- A Red Black tree is a type of self-balancing binary search tree that obeys the following rules:
  - A node is either red or black
  - The root is black
  - All leaves (NIL) are black
  - If a node is red, then both its children are black
  - Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes

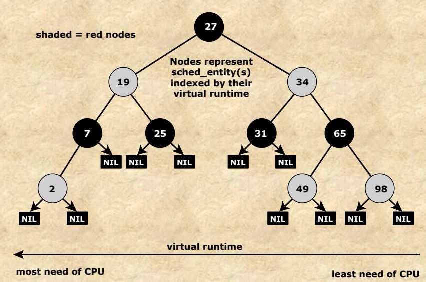


Figure 10.11 Example of Red Black Tree for CFS

UNIX SVR4 Scheduling

- A complete overhaul of the scheduling algorithm used in earlier UNIX systems

The new algorithm is designed to give:

- Highest preference to real-time processes
- Next-highest preference to kernel-mode processes
- Lowest preference to other user-mode processes

- Major modifications:
  - Addition of a preemptable static priority scheduler and the introduction of a set of 160 priority levels divided into three priority classes
  - Insertion of preemption points

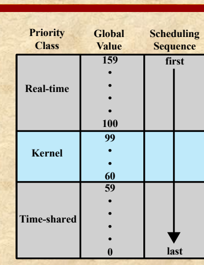


Figure 10.12 SVR4 Dispatch Queues

SVR Priority Classes

Real time (159 – 100)
- Guaranteed to be selected to run before any kernel or time-sharing process
- Can make use of preemption points to preempt kernel processes and user processes

Kernel (99 – 60)
- Guaranteed to be selected to run before any time-sharing process, but must defer to real-time processes

Time-shared (59-0)
- Lowest-priority processes, intended for user applications other than real-time applications

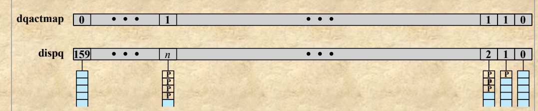


Figure 10.13 SVR Priority Classes

Table 10.6 FreeBSD Thread Scheduling Classes

Priority Class	Thread Type	Description
0 - 63	Bottom-half kernel	Scheduled by interrupts. Can block to await a resource.
64 - 127	Top-half kernel	Runs until blocked or done. Can block to await a resource.
128 - 159	Real-time user	Allowed to run until blocked or until a higher priority thread becomes available. Preemptive scheduling.
160 - 223	Time-sharing user	Adjusts priorities based on processor usage.
224 - 255	Idle user	Only run when there are no time sharing or real-time threads to run.

Note: Lower number corresponds to higher priority

SMP and Multicore Support

- FreeBSD scheduler was designed to provide effective scheduling for a SMP or multicore system
- Design goals:
  - Address the need for processor affinity in SMP and multicore systems
    - *Processor affinity* – a scheduler that only migrates a thread when necessary to avoid having an idle processor
  - Provide better support for multithreading on multicore systems
  - Improve the performance of the scheduling algorithm so that it is no longer a function of the number of threads in the system

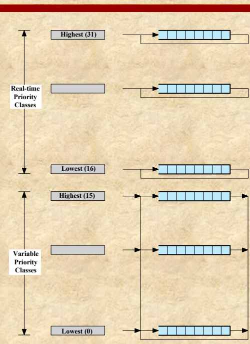


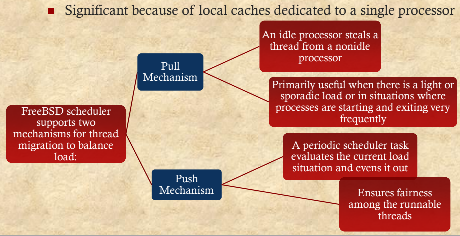
Figure 10.14 Windows Thread Dispatching Priorities

Interactivity Scoring

- A thread is considered to be *interactive* if the ratio of its voluntary sleep time versus its runtime is below a certain threshold
- Interactivity threshold is defined in the scheduler code and is not configurable
- Threads whose sleep time exceeds their run time score in the lower half of the range of interactivity scores
- Threads whose run time exceeds their sleep time score in the upper half of the range of interactivity scores

Thread Migration

- *Processor affinity* is when a Ready thread is scheduled onto the last processor that it ran on
  - Significant because of local caches dedicated to a single processor



Windows Scheduling

- Priorities in Windows are organized into two bands or classes:

Real time priority class
- All threads have a fixed priority that never changes
- All of the active threads at a given priority level are in a round-robin queue

Variable priority class
- A thread's priority begins an initial priority value and then may be temporarily boosted during the thread's lifetime

- Each band consists of 16 priority levels
- Threads requiring immediate attention are in the real-time class
  - Include functions such as communications and real-time tasks

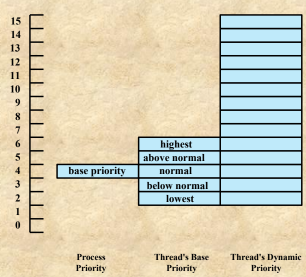


Figure 10.15 Example of Windows Priority Relationship

Multiprocessor Scheduling

- Windows supports multiprocessor and multicore hardware configurations
- The threads of any process can run on any processor
- In the absence of affinity restrictions the kernel dispatcher assigns a ready thread to the next available processor
- Multiple threads from the same process can be executing simultaneously on multiple processors
- Soft affinity
  - Used as a default by the kernel dispatcher
  - The dispatcher tries to assign a ready thread to the same processor it last ran on
- Hard affinity
  - Application restricts its thread execution only to certain processors
- If a thread is ready to execute but the only available processors are not in its processor affinity set, then the thread is forced to wait, and the kernel schedules the next available thread

Summary

- Multiprocessor and multicore scheduling
  - Granularity
  - Design issues
  - Process scheduling
  - Thread scheduling
  - Multicore thread scheduling
- Linux scheduling
  - Real-time scheduling
  - Non-real-time scheduling
- UNIX FreeBSD scheduling
  - Priority classes
  - SMP and multicore support
- Real-time scheduling
  - Background
  - Characteristics of real-time operating systems
  - Real-time scheduling
  - Deadline scheduling
  - Rate monotonic scheduling
  - Priority inversion
- Windows scheduling
  - Process and thread priorities
  - Multiprocessor scheduling
- UNIX SVR4 scheduling